Cache算法设计与实现

1. **总体思路**

在本次实验要求中，给出了Cache大小（256KB），Block大小（16B），内存地址的位数为64位（数据从外部文件中逐行读取），Cache的替换算法为LRU算法，需要分别使用直接映射DM、组关联SA方法、MRU预测方法和Multi-column预测方法模拟Cache行为，并且计算对应的命中率。在本算法设计中，MRU和Multi-column都属于提高Cache的First Hit，加快Cache命中的预测算法，最终的Cache替换策略都为LRU，因此并不会改变Cache总的命中率。

本次实验使用C++语言在VSCODE编译环境中进行算法实现，主要设计思路如图1所示：

图1 Cache算法设计总体思路

主程序（Main Process）负责从外部读取TRACE文件，解析地址中的Tag、Set与Offset, 并与Cache中的组号进行映射（b = B mod C，其中b为cache中块号，B为主存中的块号，C为cache中块容量），然后选择调用LRU、MRU或Multi-column算法对CACHE进行访问或替换，同时将命中结果输出到终端，将Miss的记录保存到外部文件中。

LRU、MRU和Multi-column算法可以封装成为一个类（CloveCache类），用于对一块地址空间进行访问策略和换入换出管理。类的底层使用LinkList、Map、Vector等数据结构进行组织，便于完成不同的功能。

Cache使用数组进行模拟。每个数组单元代表Cache中的一个组，组内采用类（CloveCache类）进行管理。CloveCache类可以完成Cache组内数据的MRU预测访问、 Multi-column预测访问以及LRU换入换出等管理工作。

1. **算法设计**
2. Cache设计

Cache采用数组进行模拟。为了设计的统一，将直接映射认为是1路组相联。设Cache大小为CACHE\_SIZE，块大小为BLOCK\_SIZE，Cache组数为CACHE\_SET，Cache组内路数为CACHE\_WAY，对于Cache大小为256KB，Block大小为16B，内存地址的位数为64位来说，则有：

const int MEM\_ADDR\_LEN = 64;

const int CACHE\_SIZE = 256 \* 1024;

const int BLOCK\_SIZE = 16;

const int CACHE\_WAY = 4; //1: DM 2: 2-way 4: 4-way……

const int CACHE\_SET = CACHE\_SIZE / BLOCK\_SIZE / CACHE\_WAY; //e.g. 2^14

//for a 64-bit memory addr, for it's block size is 16B, so offset is:4

const int offset = log(BLOCK\_SIZE)/log(2);

const int index = log(CACHE\_SET)/log(2);// the cache lines, e.g. 14

const int preOffset = log(CACHE\_WAY)/log(2);

const int tag = MEM\_ADDR\_LEN - index - offset; // e.g. 64- 14 - 4

Cache组内用CloveCache类结构，则Cache表示为：

CloveCache\* cache[CACHE\_SET]

Cache每个组的初始化需要传递组内的相联路数，表示如下：

cache[i] = new CloveCache(CACHE\_WAY)

1. CloveCache类

该类用于对Cache中的一个组内部数据结构进行定义与管理。如图2所示：



图2 Cache组内数据结构

CloveCache类使用链表结构表示LUR队列，其中Head表示mru位置，Tail表示lru位置，即新数据从Head挂入，将要被替换出去的数据从Tail进行移除。Cache中每一路采用双向链表存储地址的Tag信息，双向链表ListNode设计如下：

typedef struct ListNode

{

    ulong64 value;

    struct ListNode \*pPre;

    struct ListNode \*pNext;

} ListNode;

为了加快索引，CloveCache类使用Map结构模拟Cache Set内Ways的存储，Multi-column算法中使用Vector向量存储，对于MRU预测算法，采用一个整型值进行模拟，数据结构定义如下：

int m\_capacity; //capacity of LRU

ListNode \*m\_pHead; //head pointer, store the newest data

ListNode \*m\_pTail; //tail pointer, the LRU data

map<int, ListNode \*> m\_mp;

int m\_MRUPre; // mru for predicting, store the cache-way index

vector<vector<int>> m\_selectedLocation;

CloveCache类中定义了LRU算法、MRU算法和Multi-column的Full indexing以及Partial indexing方法。

int LRU(ulong64 value);

int MruPre(ulong64 value);

int MccPre(ulong64 value, bool beFullIndex = true);

1. LRU算法

LRU的算法流程如图3所示：



图3 LRU算法设计流程图

数据Value1到来后，首先在Cache的Map中查找数据是否存在，如果存在则直接命中，如果不存在，则根据Map当前容量的大小选择是否需要替换一块最近最少使用的数据出去。

针对Map+LinkList的数据结构，LRU在进行换入换出以及调整MRU（注意：是该节点的链表属性，不是Map中的位置）时主要进行三个操作：Tail移除操作（Remove）、Head插入操作（SetHead）以及Map赋值与记录操作。

**1．Remove方法**

Remove方法是将链表中特定Node从节点中移除，但并不删除该数据。主要有3种情况：

1）如果是头节点head，则将head移除，head的next成为head;

m\_pHead = pNode->pNext;

2）如果是尾节点tail，则将tail移除，tail的pre成为tail;

m\_pTail = pNode->pPre;

3）如果是中间某节点pnode，则将pnode移除，pnode的pre和next进行建链。

pNode->pPre->pNext = pNode->pNext;

pNode->pNext->pPre = pNode->pPre;

**2．SetHead方法**

SetHead方法直接将指定节点插入链表中Head的前面，并根据LinkList中Head和Tail的属性尽心指针定位。主要的代码如下：

pNode->pNext = m\_pHead;pNode->pPre = NULL;

m\_pHead->pPre = pNode;m\_pHead = pNode;

因此LRU中的换入换出以及调整MRU位置只可能是下面三种情况：

1）命中。将当前命中位置设置为MRU位置。

ListNode \*Node = it->second;

Remove(Node);SetHead(Node);

mruKey = it->first; // record the most resent access cache-way

2）缺失且链表满。将LRU位置（LinkList中Tail位置）数据移除，并在链表Head插入新的数据节点。此时由于是新换入了数据，因此需要同步修改map中对应位置的值。

it = find\_if(m\_mp.begin(), m\_mp.end(), MapValueFinder(m\_pTail->value));

Remove(m\_pTail);SetHead(NewNode);

m\_mp[it->first] = NewNode; //new data take place the old data

mruKey = it->first;

3）缺失且链表未满。只需在map中空位置处插入新的节点，并将该节点置为MRU位置。

SetHead(NewNode);

mruKey = m\_index; // record the most resent access cache-way

m\_mp[m\_index++] = NewNode;

1. MRU算法

MRU预测算法在首次访问Cache即命中的情况下能够加快Cache的访问时间，它是在LRU的基础上设定一个MRU变量用于存储最近最新访问的数据位置。算法主要点在于每次命中和换入换出时记录当前的最新位置，并将其设置为MRU位置。

mruKey = it->first; 或者 mruKey = m\_index;

每次从Map中查找数据前，都先检查MRU中对应的位置，如果刚好命中，则返回First Hit（return 1），不用更新MRU数值（因为命中的本来就是MRU），否则使用LRU算法。

if (it != m\_mp.end() && m\_mp[m\_MRUPre]->value == value)

return 1;

return LRU(value);

MRU的算法流程如图4所示：



图4 MRU算法设计流程图

1. Multi-column算法

相比较与MRU，Multi-column能够加大first hits的命中率，并且减少non-first-hits的时间延迟，在first hit命中时相当于一次直接映射，在增加少量硬件的情况下这是非常值得和划算的。本次算法模拟的主要思想是增加一个二维向量用于模拟Selected Locations的硬件结构，算法设计主要考虑以下方面：

1）使用Tag的最后n位（n= log2(CACHE\_WAY)）直接对应Cache中的Major Location。Major Location是由Tag的最后n位决定的，从Mem中新调度的数据以及刚刚访问过的数据都应该放到对用的Major Location位置上；

2）First hit通过map的key索引直接访问数据，相当于模拟一次直接映射；

3）Multiple blocks can be direct -mapped to the same major location, but only MRU is the major；

4）The non-MRU blocks are stored in other empty MRU blocks are stored in other locations in the set: Selected Locations；

5）If “other locations” are used for their own major locations, there will be no space for selected ones；

6）Swap: A block in selected selected location is swapped to major location after a new block is loaded in from memory。

Multi-column使用Full indexing和Partial indexing两种策略。相比于Partial indexing，Full indexing是将所有对应的非Major Location数据都在Selected Locations中进行标记（标记为1），优势在于所有Tag末尾n位相同的数据，如果还在Cache里，肯定能通过Selected Locations快速Hit，从而加快命中速度；而Partial indexing的优势在于，only tracing one selected location，这样可以减少硬件的逻辑以及降低复杂度，而两次命中检测基本已经可以大大加快命中速度，所以是一种tradeoff，各有利弊。

Multi-column的主算法流程（包括Full indexing和Partial indexing）如图5所示：



图5 Multi-column主算法设计流程图

算法流程描述如下：

1）根据Cache组内路数的大小获取Multi-column的Major Location值；

int predictedWay = value1 % m\_capacity;

2）根据predictedWay（PreOffset）值直接映射到Map中匹配Major Location，如果该位置为空，则将value放置在当前位置，设置该位置为MRU，并在Selected Locations中设置主位置标志为-1（用-1标志主位置，便于后面算法进行swap），返回miss(0)；如果不为空，则转向3）；

       m\_selectedLocation[predictedWay][predictedWay] = -1;

3）如果是First hit，则将当前位置设置为MRU，并返回first hit(1)；否则转向4；

4）如果是Full indexing，则转向5），Partial indexing则转向9）；

if (beFullIndex)

   return SearchSelectedLocationsFullIndexing(value, predictedWay);

return SearchSelectedLocationsPartialIndexing(value, predictedWay);

5）根据Selected Locations进行non-first-hit，只需要检查值为1的位置，如果命中，则设置该位置为MRU，并将它和Major Location互换，返回2；否则转向6）；

if (m\_selectedLocation[predictedWay][i] == 1) // selected location i

   if (m\_mp[i] && m\_mp[i]->value == value) //hit

   { ListNode \*Node = m\_mp[i];

       Remove(Node); SetHead(Node);  Swap(i, predictedWay);

return 2;

}

}

6）判断当前Map是否已满，如果未满，则转向7；如果已满，则首先根据LRU策略寻找value1要替换的位置，并将该节点（是该节点的链表属性，不是Map中的位置）设置为MRU，然后将该节点与Major Location中的节点互换位置。互换位置前需要将自己所在Way以及其他Way中的Selected Locations标记进行改变，遵循的原则如下：

1. 如果是自己所在的Way，则将Major Location在Selected Locations中的标志取反设置到Value当前所在位置的Selected Locations中，并将Major Location位重新设置为-1。这是因为如果原来Major Location位置的标记为0，则说明该位置已被其他数据占据，互换位置后对应位的标记也应该为0；如果Major Location位置的标记为-1，则说明该位置为主位置，互换位置后对应位的标记应该为1。
2. 如果不是自己所在的Way，则将Major Location在Selected Locations中的标志直接设置到value1当前所在位置的Selected Locations中（告诉对方已将其Selected Location移到了其他位置上），并将Major Location位重新设置为0（告诉对方该位置我已收回）；

if (i != predictedWay)

{

   m\_selectedLoc[i][it->first] = m\_selectedLoc[i][predictedWay];

   m\_selectedLoc[i][predictedWay] = 0;

}

else

{

   m\_selectedLoc[i][it->first] = -m\_selectedLoc[i][predictedWay];

   m\_selectedLoc[i][predictedWay] = -1;

}

7）寻找空位置放置该value1，并将该节点设置为MRU，然后将该节点与Major Location中的节点互换位置。遵循的原则与6）相同；

8）6）或7）完成后返回0。

9）根据Selected Locations进行non-first-hit，只需要检查值为1的位置，并且只检查一次（一个Major Location和一个Selected Location已经能够大程度上加大first hits的命中率，并且减少non-first-hits的时间延迟，是一种tradeoff策略）。如果命中，则设置该节点为MRU，并将它和Major Location互换，返回2；否则转向10）；

10）使用LRU方法在Map中其他位置寻找匹配（此时在查询时间上不再具有优势），如果命中，则设置该节点为MRU，并将它和Major Location互换，返回3；否则转向11）；

11）该策略与6）基本一致，只在Selected Location的设置上有所不同。判断当前Map是否已满，如果未满，则转向12）。与6）的主要区别在于需要判断Selected Locations中是否已经有一个Location为1，如果已存在，则放置的新位置的Selected Location需要设置为0。完成后返回0。

//判断是不是已经有一个selected pos了,或者当前替换出去的就是这个标识为1的位置

if(selectedPos < 0 || it->first == selectedPos)

{

  m\_selectedLoc[i][it->first] = -m\_selectedLoc[i][predictedWay];

  m\_selectedLoc [i][predictedWay] = -1;

}

else//在其他位置已经存在一个是1的位置了

{

  m\_selectedLoc[i][it->first] = 0;

  m\_selectedLoca[i][predictedWay] = -1;//将自己的MRU位置标记为-1

}

12）寻找空位置放置该value1，并将该节点设置为MRU，然后将该节点与Major Location中的节点互换位置。遵循的原则与11同。完成后返回0。

1. **实验结果**

实验主要完成3个目标：

1）Cache预测方法对命中率的影响。主要分析LRU、MRU和Multi-column在2-way、4-way、8-way、16-way组关联SA方法下Cache命中率的大小以及原因；

2）平均搜索长度计算。计算Multi-column的 full indexing策略在major location被其他major location group的selected location占用时，其对bit vector的平均搜索长度。

3）Cache块大小对命中率的影响。分析使用组关联SA方法组织Cache，且使用4-way时，当块大小分别为8 Bytes、16 Bytes、32Bytes、64Bytes、128Bytes、256Bytes以及512Bytes时Cache命中率的大小以及原因。

1. Cache预测方法对命中率的影响

以4路组相联为例，实验运行结果如图6所示：

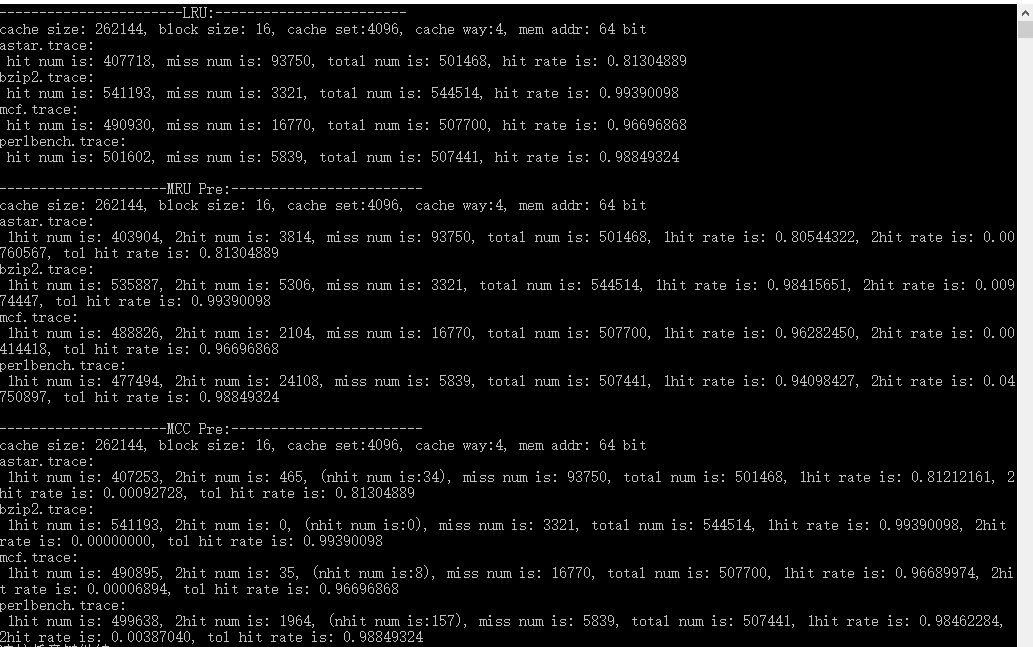


图6 4路组相联下实验运行结果

实验结果统计如表1所示：

表1 LRU、MRU和Multi-column在特定条件下的实验结果

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **实验条件：**   * Cache size: 256 KB. * Block size: 16 Bytes. * For a 64-bit memory address. * Cache replacement algorithm: LRU. * Cache way: DM\2-way\4-way\8-way\16-way * astar.trace: 501468 lines. * bzip2.trace: 544514 lines. * mcf.trace: 507700 lines. * perlbench.trace: 507441 lines.   **数据说明：**   * 为了标识出数据的差异性，命中率统一采用双精度浮点数，精确到小数点后8位。 * 对于Multi-column的Partial indexing列，每个表格内第一行表示Non-first Hit Rate,第二行在括号内的浮点数表示不通过Selected Locations查找的命中率。 | | | | | | | | | |
|  | | | **First Hit**  **(Rate)** | **Non-first Hit**  **(Rate)** | | | | **Miss**  **(Cnt)** | **Total Hit**  **(Rate)** |
| **astar** | **LRU** | **DM** | - | - | | | | 94214 | 0.81212361 |
| **SA 2** | - | - | | | | 93807 | 0.81293522 |
| **SA 4** | - | - | | | | 93750 | 0.81304889 |
| **SA 8** | - | - | | | | 93741 | 0.81306684 |
| **SA 16** | - | - | | | | 93741 | 0.81306684 |
| **MRU** | **SA 2** | 0.81054624 | 0.00238899 | | | | 93807 | 0.81293522 |
| **SA 4** | 0.80544322 | 0.00760567 | | | | 93750 | 0.81304889 |
| **SA 8** | 0.80342514 | 0.00964169 | | | | 93741 | 0.81306684 |
| **SA 16** | 0.80040202 | 0.01266482 | | | | 93741 | 0.81306684 |
| **Multi-**  **column** |  |  | Full  indexing | | | Partial  indexing |  |  |
| **SA 2** | 0.81205780 | 0.00087742 | | | 0.00087742  (0) | 93807 | 0.81293522 |
| **SA 4** | 0.81212161 | 0.00092728 | | | 0.00092728  (0.00006780) | 93750 | 0.81304889 |
| **SA 8** | 0.81212361 | 0.00094323 | | | 0.00094323  (0.00013959) | 93741 | 0.81306684 |
| **SA 16** | 0.81212361 | 0.00094323 | | | 0.00094323  (0.00013959) | 93741 | 0.81306684 |
| **bzip2** | **LRU** | **DM** | - | - | | | | 3321 | 0.99390098 |
| **SA 2** | - | - | | | | 3321 | 0.99390098 |
| **SA 4** | - | - | | | | 3321 | 0.99390098 |
| **SA 8** | - | - | | | | 3321 | 0.99390098 |
| **SA 16** | - | - | | | | 3321 | 0.99390098 |
| **MRU** | **SA 2** | 0.98483786 | 0.00906313 | | | | 3321 | 0.99390098 |
| **SA 4** | 0.98415651 | 0.00974447 | | | | 3321 | 0.99390098 |
| **SA 8** | 0.98341640 | 0.01048458 | | | | 3321 | 0.99390098 |
| **SA 16** | 0.97845969 | 0.01544129 | | | | 3321 | 0.99390098 |
| **Multi-**  **column** |  |  | Full  indexing | | Partial  indexing | |  |  |
| **SA 2** | 0.99390098 | 0 | | 0 | | 3321 | 0.99390098 |
| **SA 4** | 0.99390098 | 0 | | 0 | | 3321 | 0.99390098 |
| **SA 8** | 0.99390098 | 0 | | 0 | | 3321 | 0.99390098 |
| **SA 16** | 0.99390098 | 0 | | 0 | | 3321 | 0.99390098 |
| **mcf** | **LRU** | **DM** | - | - | | | | 16805 | 0.96689974 |
| **SA 2** | - | - | | | | 16772 | 0.96696474 |
| **SA 4** | - | - | | | | 16770 | 0.96696868 |
| **SA 8** | - | - | | | | 16770 | 0.96696868 |
| **SA 16** | - | - | | | | 16770 | 0.96696868 |
| **MRU** | **SA 2** | 0.96327950 | 0.00368525 | | | | 16772 | 0.96696474 |
| **SA 4** | 0.96282450 | 0.00414418 | | | | 16770 | 0.96696868 |
| **SA 8** | 0.95865078 | 0.00831790 | | | | 16770 | 0.96696868 |
| **SA 16** | 0.94315147 | 0.02381721 | | | | 16770 | 0.96696868 |
| **Multi-**  **column** |  |  | Full  indexing | Partial  indexing | | |  |  |
| **SA 2** | 0.96689777 | 0.00006697 | 0.00006697  （0） | | | 16772 | 0.96696474 |
| **SA 4** | 0.96689974 | 0.00006894 | 0.00006894  （0.00001576） | | | 16770 | 0.96696868 |
| **SA 8** | 0.96689974 | 0.00006894 | 0.00006894  （0.00001576） | | | 16770 | 0.96696868 |
| **SA 16** | 0.96689974 | 0.00006894 | 0.00006894  （0.00001576） | | | 16770 | 0.96696868 |
| **perlbench** | **LRU** | **DM** | - | - | | | | 7756 | 0.98471546 |
| **SA 2** | - | - | | | | 6297 | 0.98759068 |
| **SA 4** | - | - | | | | 5839 | 0.98849324 |
| **SA 8** | - | - | | | | 5752 | 0.98866469 |
| **SA 16** | - | - | | | | 5752 | 0.98866469 |
| **MRU** | **SA 2** | 0.97018767 | 0.01740301 | | | | 6297 | 0.98759068 |
| **SA 4** | 0.94098427 | 0.04750897 | | | | 5839 | 0.98849324 |
| **SA 8** | 0.91641590 | 0.07224879 | | | | 5752 | 0.98866469 |
| **SA 16** | 0.89293534 | 0.09572936 | | | | 5752 | 0.98866469 |
| **Multi-**  **column** |  |  | Full  indexing | Partial  indexing | | |  |  |
| **SA 2** | 0.98429965 | 0.00329102 | 0.00329102  （0） | | | 6297 | 0.98759068 |
| **SA 4** | 0.98462284 | 0.00387040 | 0.00387040  （0.00030940） | | | 5839 | 0.98849324 |
| **SA 8** | 0.98471546 | 0.00394923 | 0.00394923  (0.00024436) | | | 5752 | 0.98866469 |
| **SA 16** | 0.98471546 | 0.00394923 | 0.00394923  (0.00024436) | | | 5752 | 0.98866469 |

从实验结果可以看出：

1）对于LRU方法，当路数增加时，总的命中率呈递增上升趋势，并最终逼近一个稳定值，这个值应该是全相联的命中率。

2）随着路数的增加，几类trace的命中率无明显提升，特别当从8路到16路，命中率没有提升。这是因为根据局部性原理，trace文件中同一数据的访问跨度已远远大于路数的增加，其局部性原理相对较差或者可以说是long distance reuse，因此整体命中率不会随着路数的增加而增加，也就是说当mem中数据的随机性比较大的时候，增加路数已不能解决命中率的问题，需要采用其他更加优化的方法进行命中率提升。

3）使用MRU预测方法的优点是如果命中，可以有效减少access time，即使首次命中失败，也可以按传统方法检查其他位置。从实验结果可以看出，随着路数的增加，MRU的first hit在逐渐减少（如图7所示），这是因为随着路数的增大，放在同一个组内的数据将增多，MRU位置将会有更大的数据基数与数据随机性，从而会降低首次命中的概率。但是由于MRU也采用LRU替换方法，因此总的命中率与LRU相同。

4）MRU的缺点是它的first hit是在关键查找路径上，由于它采用额外的存储空间，在查找路径上需要先获取MRU index，因此它的first hit并不等同于直接映射的命中；而且由于它只采用一位MRU location，因此对于long reuse distance的数据来说，命中率会大大降低。

5）相比于MRU，Multi-column由于直接使用Tag中的后n位作为major lacation检索，因此first hit等同于一次直接命中，可大幅度减少access time。从实验结果可以看出，Multi-column的first hit rate比MRU要高很多（如图7所示），且随着路数的增加，first hit rate也相应的增加，并逐渐逼近总的命中率，这是因为随着路数的增加，major lacation位也相应的增加，从而提高first hit rate。

6）由于具有较高的关联性，Multi-column的命中率高于直接映射的命中率，但又在很高的概率上具备直接映射的低访问延迟，而且由于Selected Locations的帮助，它的non-first hit命中延迟也要比SA快的多。

7）从Multi-column的 Partial indexing结果可以看出，采用一位 selected location就已经可以达到很高的命中率，因此在一定程度上tradeoff是会有比较好的相对收益。



图7 First hit rate随路数大小变化曲线（astar）

1. 平均搜索长度计算

实验结果统计如表2所示：

表2 Multi-column在 full indexing下major location被

其他major location group的selected location占用时，其对bit vector的平均搜索长度

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **实验条件：**   * Cache size: 256 KB. * Block size: 16 Bytes. * For a 64-bit memory address. * Cache algorithm: Multi-column. * Cache way: 2-way\4-way\8-way\16-way * astar.trace: 501468 lines. * bzip2.trace: 544514 lines. * mcf.trace: 507700 lines. * perlbench.trace: 507441 lines. | | | | |
|  | **2-way** | **4-way** | **8-way** | **16-way** |
| **astar** | 0 | 0 | 0 | 0 |
| **bzip2** | 0 | 0 | 0 | 0 |
| **mcf** | 0 | 0 | 0 | 0 |
| **perlbench** | 0 | 0 | 0 | 0 |

1. Cache块大小对命中率的影响

实验结果统计如表3所示：

表3 不同块大小在4路组相联下实验结果

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **实验条件：**   * Cache size: 256 KB. * Block size: 8 Bytes\16 Bytes\32 Bytes\64 Bytes\128 Bytes\256 Bytes\512 Bytes. * Cache way: 4-way. * For a 64-bit memory address. * Cache replacement algorithm: LRU. * astar.trace: 501468 lines. * bzip2.trace: 544514 lines. * mcf.trace: 507700 lines. * perlbench.trace: 507441 lines.   **数据说明：**   * 为了标识出数据的差异性，命中率统一采用双精度浮点数，精确到小数点后8位。 | | | | | | | | |
|  | | **8B** | **16B** | **32B** | **64B** | **128B** | **256B** | **512B** |
| **astar** | **Hit Count** | 385412 | 407718 | 453508 | 476724 | 488604 | 494718 | 497858 |
| **Hit Rate** | 0.76856749 | 0.81304889 | 0.90436080 | 0.95065687 | 0.97434732 | 0.98653952 | 0.99280114 |
| **bzip2** | **Hit Count** | 537887 | 541193 | 542846 | 543673 | 544089 | 544298 | 544402 |
| **Hit Rate** | 0.98782951 | 0.99390098 | 0.99693672 | 0.99845550 | 0.99921949 | 0.99960332 | 0.99979431 |
| **mcf** | **Hit Count** | 484468 | 490930 | 498437 | 502198 | 504937 | 506305 | 506992 |
| **Hit Rate** | 0.95424069 | 0.96696868 | 0.98175497 | 0.98916289 | 0.99455781 | 0.99725231 | 0.99860548 |
| **perlbench** | **Hit Count** | 498518 | 501602 | 504002 | 505387 | 506145 | 506527 | 506482 |
| **Hit Rate** | 0.98241569 | 0.98849324 | 0.99322286 | 0.99595224 | 0.99744601 | 0.99819881 | 0.99811013 |

从实验结果可以看出：当Cache的容量一定时**，** 在采用4路组相联下，块的大小对命中率的影响非常敏感。随着块数的增加，整体呈现命中率上升趋势（如图8所示），这是因为随着块数的增加，容量缺失减少，其允许利用更多的空间局部性原理，从而导致cache有更高的命中率。

图8 命中率随块大小变化曲线

但是随着块数增加，在perlbench上出现命中率下降的现象。块数继续增加，在其他三类trace上命中率也都开始下降，原因分析如下：

假设Cache总量为 T，组数为 S，组内块数为W，块大小为 B，因此有：T=S\*W\*B。由该式可以看出，当T和W不变时，Cache 块大小与组数成反比。当Cache的块容量B很小时**，**cache的组就多，主存中的某一块可以映射到Cache中的块数就少，所以Cache 的命中率也比较低。随着块大小的增加， 由于程序的空间局部性起主要作用，同一块中数据的利用率比较高，Cache 的命中率也随之升高。但是当块的大小增大时，Cache lines也在减少，这使得具有long distance reuse数据的概率变小，这种增加趋势在某一个trade off处使 Cache 命中率达到最大值，之后命中率随着块大小的增加反而减小。而且，随着块大小的增加，程序时间局部性的作用就会逐渐减弱。所以，当块大小增加时**，**命中率开始时处于上升趋势，后来反而会下降。

1. **总结提高**

通过本次实验，我对cache的LRU算法、MRU算法以及Multi-column有了深刻的理解，并且通过自己模拟cache，对cache的结构以及cache直接映射、组相联映射和全相联映射有了进一步理解。通过研究这些算法和完成本次实验，对自己在以后的创新实践和科研工作中也会有更多的帮助，后续希望通过自己多动手做模拟实验，对高等计算机系统结构的存、算、传有更深刻的理解，从而更好的完成自己的毕业课题和设计。感谢张老师的解惑以及杨青霖助教的指导，感谢实验过程中李之悦同学和魏钧宇学长的帮助。

1. **参考文献**

[1] Zhang, et. al., Multi-column Caches: Fast Accesses and High Hit Ratio, IEEE Micro, 1997.